15-122 : Principles of Imperative Computation, Fall 2015

Written Homework 1

Due: Tuesday, September 8, 2015

Name: ______________________________

Andrew ID: ______________________________

Section: ______________________________

This written homework will introduce you to the way we reason about C0 code in 15-122.

Print out this PDF double-sided, staple pages in order,
and write your answers *neatly* by hand.

The assignment is due at the start of your lecture on Tuesday,
September 8, 2015.

Score Table (for course staff only):

Question	Points	Score
1	2	
2	7	
3	6	
Total:	15	

1. **Running C0 programs**

   The file `foo.c0` contains a function `foo` that takes an integer argument and returns an integer. Additionally, there is a `foo-test.c0` file that contains the following:

```
/* 1 */  int main() {
/* 2 */      return foo(15234);
/* 3 */  }
```

(1) (a) From the command line, show how to display the value returned by `foo(15234)` using the C0 compiler.

Solution:

(1) (b) From the command line, show how to display the value returned by `foo(15234)` using the C0 interpreter.

Solution:

--> foo(15234);

2. **Using contracts**

   In this question, we'll examine a function `compute_sum`, which computes the sum of the first m positive integers, where m is a non-negative integer:

```
int compute_sum(int m) {
   int total = 0;
   while (m > 0) {
      total = total + m;
      m = m - 1;
   }
   return total;
}
```

(1) (a) Complete the specification function below that we will use to help us prove the correctness of the function above. The `SUM` function should return the sum of the first m positive integers using a simple mathematical formula instead of a loop or a recursive call:

Solution:

```
/*  1 */  int SUM(int m)
/*  2 */
/*  3 */  //@requires m > 0;
/*  4 */  {
/*  5 */       return (m+1) * m / 2;
/*  6 */  }
```

(1) (b) Given the specification function above, give a suitable precondition and descriptive postcondition in C0 for the `compute_sum` function. You may assume that `num_ints` is less than 2^{15}. (You'll see why this is important next week.)

Solution:

```
/*  7 */  int compute_sum(int num_ints)

/*  8 */  //@requires m > 0;

/*  9 */  //@ensures \result == SUM(num_ints);
/* 10 */  {
/* 11 */      int m = num_ints;
/* 12 */      int total = 0;
/* 13 */      while (m > 0)
/* 14 */      //@loop_invariant m >= 0;
/* 15 */      // Additional loop invariant will go here (part d)
/* 16 */      {
/* 17 */          total = total + m;
/* 18 */          m = m - 1;
/* 19 */      }
/* 20 */      //@assert m <= 0;
/* 21 */      return total;
/* 22 */  }
```

An aside: in the example above, we're using a specification function that's ***more*** *efficient than the function we're analyzing. Unlike in the example from lecture, it's much better to compute sums the way we do in part (a) – in the specification function – than the way we do it in part (b) – in the implementation. We're just using the more efficient function to test our correctness in the less efficient function.*

(1) (c) Explain why we introduced the variable `num_ints` in part (b).

Solution:
Ensures SUM() won't be confused.

(1) (d) Why would `m > 0` on line 14 have been an invalid loop invariant?

Solution:
Because loop-invariant is checked before exiting the loop. In the last loop, m = m - 1 would become 0.

(1) (e) Give a suitable loop invariant on the variable `total` that will help us prove that the postcondition is satisfied.

Solution:

```
/* 15 */  //@loop_invariant total + SUM(m) == SUM(num_ints);
```

(1) (f) Assuming that the loop invariants are correct and the precondition is satisfied, show that the postcondition must hold.

Solution:

```
[Assertion 1] ---
Line 14 and Line 20 implies:
```
m = 0

```
[Assertion 2] ---
Line 15 and [Assertion 1] implies:
```
m == 0; total == SUM(num_ints);

(1) (g) The reasoning above assumes that the loop terminates. Looking at the body of the loop, concisely explain why the loop must terminate if the precondition is satisfied. (Think about what quantity decreases with each iteration but cannot go negative.)

Solution:
m is positive when enter the loop;
m - 1 < m means m is strictly decreasing in the loop

=> m must eventually becomes zero.

3. **Reasoning with Loop Invariants**

Consider the following function in C0:

```
/*   1 */  int mystery1(int n)
/*   2 */  //@requires n > 0;
/*   3 */  //@ensures \result == n * n;
/*   4 */  {
/*   5 */      int k = 0;
/*   6 */      int i = 0;
/*   7 */      while (i < n)
/*   8 */      //@loop_invariant 0 <= i && i <= n;    /* LI1 */
/*   9 */      //@loop_invariant k == i * i;          /* LI2 */
/*  10 */      {
....           // body of loop, which modifies k and i, not shown
/* y-2 */      }
/* y-1 */      return k;
/*   y */  }
```

Assume that the precondition is satisfied for the function above.

(1) (a) If the loop terminates, what is true about `i` on line `y-1` based on the loop condition on line 7 immediately after the loop guard evaluates to `false`?

Solution: i >= n;

Given your observation above and the first loop invariant on line 8, what can you infer about the value of `i` immediately after the loop guard evaluates to `false`?

Solution: i == n;

(1) (b) Given the final result in part (a) and the second loop invariant on line 9, what can you conclude about the value of the variable `k` immediately after the loop guard evaluates to `false`?

Solution: k == n * n;

How do your observations about `k` above prove that the postcondition is satisfied?

Solution:

(2) (c) Here is a loop body for a *different* function – the loop invariants are the same, but the loop guard and returned value are different. A function body has also been included.

```
/*  1 */  int mystery2(int n)
/*  2 */  //@requires n > 0;
/*  3 */  // postcondition not shown
/*  4 */  {
/*  5 */      int k = 0;
/*  6 */      int i = 0;
/*  7 */      while (k < n)
/*  8 */      //@loop_invariant 0 <= i && i <= n;    /* LI1 */
/*  9 */      //@loop_invariant k == i * i;          /* LI2 */
/* 10 */      {
/* 11 */          k = k + 2*i + 1;
/* 12 */          i = i + 1;
/* 13 */      }
/* 14 */      return i;
/* 15 */  }
```

For an initial n of 47, trace the values of `k`, `i`, and `n`, always reporting their values immediately *before* the loop guard is checked. Observe that the loop invariants should be true each time.

Solution:

Iteration	k	i	n	LI1	LI2
0 (initial)	0	0	47	true	true
1	1	1	47	true	true
	4	2	47	true	true
	9	3	47	true	true
	16	4	47	true	true
	25	5	47	true	true
	36	6	47	true	true
	49	7	47	false	true

For an initial n of 16 (in the function from part(c)), trace the values of `k`, `i`, and `n`, always reporting their values immediately *before* the loop guard is checked.

Solution:

Iteration	k	i	n	LI1	LI2
0 (initial)	0	0	16	true	true
1					
	1	1	47	true	true
	4	2	47	true	true
	9	3	47	true	true
	16	4	47	true	true

(1) (d) In one sentence, describe what this function in part (c) is computing. (What is its postcondition?)

Solution: Find the biggest square root i that is i*i < n;

i*i < n && (i+1)*(i+1) > n;

(1) (e) *Harder:* what additional loop invariant(s) would be necessary in the function in part (c) to prove that the postcondition you gave holds?

Solution: